Snowflake Naive Bayes and TPC-H

Jacob Rohde (jacro@itu.dk)

December 2024

1 Introduction

Snowflake [1] is a cloud native data platform blabla. In the following we wish to investigate and performance test two implementations of Naive Bayes on Yelp Reviews, a pure SQL implementation and a UDTF. We will briefly discuss performance and ease of implementation.

Additionally we will conduct a standard TPC-H performance experiment with different configurations of Snowflake.

2 Methodology

2.1 Naive Bayes

The provided Yelp Review dataset contains 5 labels between 0 and 4, presumably representing a rating with 0 being worst and 4 being best.

I load the test and training datasets into my database using SnowSQL windows client. From thereon after I use worksheets in the web interface SnowSight.

I create a transformed table, `yelp_review_training` and `yelp_review_testing`, for both datasets with columns id INT, label INT and text STRING. I load the text column by transforming the text field of the jsons using regex in the following manner: convert to lower space, convert non-alphanumeric to space, add space between letter-number transitions and remove extra spaces.

I then create a new table, `train_exploded_reviews` and `test_exploded_reviews` with columns `review_id` INT, word STRING, label INT, and populate it with data from the `yelp_review_training` using lateral flatten on the text column to create one row per word.

These exploded tables are used in both the pure SQL implementation and the UDTF implementation.

2.1.1 SQL

In the following I will first outline how I calculate the priors and the likelihoods for each label and word in training set. I will subsequently outline how I use the priors and likelihoods to predict the reviews in the test set. The SQL code can be found in the github repository under `naive_bayes/SQL/naive_bayes.sql`.

Priors and Likelihoods from the training set

I create a priors table which I populate by counting how many reviews each label has and divide it by the total number of reviews.

Remember that the likelihood is defined as

$$P(w|c) = \frac{count(w,c)+1}{\sum_{w' \in V} count(w',c) + |V|}$$

where

$P(w|c)$ is the probability of word w given class c,
$count(w, c)$ is the frequency of word w in class c,
V is the vocabulary (set of all unique words),
$|V|$ is the size of the vocabulary,
The summation $\sum_{w' \in V} count(w', c)$ represents the total word count in class c.

To create a likelihoods table, I first create a table `word_freq_pr_class`. I use a CTE that cross joins every distinct word in the training set with every label. I then left join onto the word-label combinations every word in the corpus with join-predicate on word and label. I group by word and label, so I can use the aggregate function count to get the frequency of each word per label. Hence the name of the table.

I then create the likelihoods table by summing the number of words per class in a CTE called `freq_pr_class`. I can then divide the word frequency by class with the total frequency by class plus the vocabulary size.

Evaluation

I create a table `review_probabilities` which I populate with the true label and the predicted label for the test set, using the likelihoods and priors obtained from the training set. By constructing such a table different metrics can always be retrieved afterwards. In the following I will go over CTEs and final select query used in populating the table.

CTEs

- *review_probabilities*: For each review and possible class label combination, it calculates the sum of log likelihoods `SUM(LN(COALESCE(l.likelihood, 1e-10)))`. Takes natural log of each word's likelihood, and uses 1e-10 if likelihood is NULL (word not seen in training). It then sums these logs, which is equivalent to multiplying probabilities. It then gets log of prior probability: `LN(p.prior)` and uses `CROSS JOIN` to ensure every review is tested against every possible label.
- *final_predictions*: Combines the log likelihood and log prior: `log_likelihood + log_prior`, which is equivalent to applying Naive Bayes in log space. It then converts back to regular probability space using the `EXP` function.
- *best_predictions*: Uses `ROW_NUMBER()` to rank predictions for each review. It orders by probability DESC so rank 1 is the highest probability. It keeps track of both `true_label` and `predicted_label` for evaluation.

The final SELECT returns `review_id`: which review we're predicting, `true_label`: the actual label from the test set, `predicted_label`: the model's prediction `probability`: how confident the model is in this prediction

This implements Naive Bayes classification in SQL in log space to avoid numerical underflow.

Confusion Matrix Analysis
The confusion matrix is calculated using two CTEs generated by the CoPilot supplied in SnowSight. For the sake of brevity and a 4 page limit I shall not cover it here.

In the github repository under `naive_bayes/SQL/visualize_confusion_matrix.py` there is a Python script which generates the confusion matrix as a PNG, also generated by CoPilot.

2.1.2 Python UDTF

My Python UDTF code can be seen in three scripts under the directory `naive_bayes/UDTF/` - three of the scripts names indicate that they were failed attempts, and the last works.

Before I outline my UDTF I will first comment on the nature of Snowflake UDTFs. I have not found in the official Snowflake documentation any mention of ML-use cases for UDTFs, but rather UDTFs seem to be used for writing custom aggregate functions or 1-1 mappings. UDTFs are executed over partitions of the data, and the state kept by the UDTF is local to each partition. We want to count across the entire training dataset, we don't want to partition our data. It is possible to chain UDTFs, which to me feels like the programmer has to fight the design of the UDTFs.

After struggling with two solutions (`NB_flat_failed.sql` and `NB_nonflat_failed.sql`) that tried to calculate likelihoods and probabilities, I decided to chain several UDTFs. I pursued an unelegant wordcounting UDTF (`NB_wordcount_failed.sql`), but abandoned it when I realized I could cheat and use "partition over 1" to force a single partition. Whether this always works or not I don't know, but it feels like it is against the rules. This solution can be found in `NB_classifier.sql`, and is outlined below.

This UDTF creates a table that has the priors, likelihoods, and frequencies - it is basically the same as the pure SQL implementation, only the tables are joined to a single table. The table has word and label as candidate key.

Joining the words in a document with the UDTF table, then grouping by label and summing the log likelihoods and adding the log prior, will yield the probabilities for each label. One then only has to select the highest prediction. I use a mechanism similar to the one described for the pure SQL implementation to populate a table of all the test dataset and then retrieve a confusion matrix.

The UDTF works by, in the process method, cleaning the input document and extracting words (tokenizing), incrementing the count for the label and for every word it adds the word to the vocabulary and updates a hashmap for the counts per word. The endPartition method then calculates the priors for each label, and the likelihoods for every label for every word, and yields the vocabulary size and word frequencies along with every row. This results in redundant information, but it is the only way I could think of that would only require a single pass over the input.

2.1.3 Execution Times

I use a large warehouse for both implementations.

For the pure SQL implementation I time the execution time of cleaning and exploding the training and testing sets, and then I time how long it takes from after cleaning and exploding until having populated the priors and likelihoods (table creation included), and I then time the evaluation of the entire test set not withoutcounting the computation of the confusion matrix. In the scripts select statements with start and end section flags can be seen to see exactly what is timed. I time it by looking at the query history manually. I do it 5 times to validate that the times I get are reasonable.

For the Python UDTF I time the execution of the UDTF which populates the table of priors and likelihoods. I then time the evaluation of the test set, not withcounting the confusion matrix computation.

This is admittedly not the most reliable way to time executions, but it will serve as a rough estimate to see which implementation is more performant. As the training consists of several queries, the Snowflake scheduler will introduce inconsistencies, further complicating a production-grade profiling.

A more thorough framework for timing executions are given for my TPC-H profiling.

2.2 TPC-H

For my TPC-H implementation I use data from the Snowflake-supplied database `SNOWFLAKE_SAMPLE_DATA`. I test TPC-H query 1, 5 and 18, which I copied from a website[1] and modified them to use Snowflake date functions.

Snowflake keeps a query history which, as everything else in Snowflake, can be queried on with SQL[2]. Every query gets hashed, and the query hash can subsequently be used to query the query history and analyze the performance of every execution of the query.

I have a Python script under my Github repository (`tpch/snowflake_tpch_generate.py`), which takes as program argument the number of iterations. It then generates a SQL script that can be copied into a SQL Worksheet in the snowflake client and executed.

[1] `https://docs.deistercloud.com/content/Databases.30/TPCH\%20Benchmark.90/Sample\%20querys.20.xml?embedded=true\#7b15827ccb57cbcec68be7a26953590c`

[2] `https://docs.snowflake.com/en/user-guide/performance-query-exploring#query-track-the-average-performance-of-a-query-over-ti`

The generation script puts a no-cache statement (`ALTER SESSION SET USE_CACHED_RESULT=FALSE;`) before each query. It then executes query 1, 5 and 18 on 4 sizes (extra small, small, medium, large) of warehouses, over scalefactor 1, 10, 100 and 1000. The warehouse is changed with the `USE WAREHOUSE` statement, and the scalefactors are changed with the `use schema` statement, (e.g. `use schema snowflake_sample_data.tpch_sf1000`).

To distinguish between the 3 queries on the different scale factor and warehouse settings, I augment the queries to select a string literal to force a unique query hash, e.g.

```
SELECT 'BISON_WH_XS_Q1_SF1000', (...)
```

I have 48 distinct queries (3 queries * 4 warehouses * 4 scalefactors). I collect the query hashes manually from the query history, and I can then analyse the execution times:

```
    SELECT
    query_parameterized_hash,
    AVG(total_elapsed_time) as mean_elapsed_time,
    ...
FROM SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY
WHERE query_parameterized_hash IN ( ... )
GROUP BY query_parameterized_hash;
```

I execute the queries 20 times across a whole day.

3 Results and Discussion

3.1 TPC-H

I have not found anything interesting in my profiling data. Indeed the data exploration I have done has been done in Python and not Snowflake, and is therefore not interesting when viewing this project as about Snowflake.

I start my analysis by investigating correlation between megabytes scanned and execution time. I find the Pearson correlation to be almost 1 for all warehouses and all queries. Given that we group by warehouse and query type, we end up checking correlation between the different scale factors and with a factor of 10 between each scale factor it is perhaps not so strange that the MB scanned end up having a strong correlation.

In figure 1 we see that query 18 scales the most linearly, whereas the others are penalized less by scale factor. Indeed Query 1 seems to have a better exeuction time for scalefactor 10 than for scalefactor 1 which is very curious, especially as it is supposed to be the most scan-intensive query.

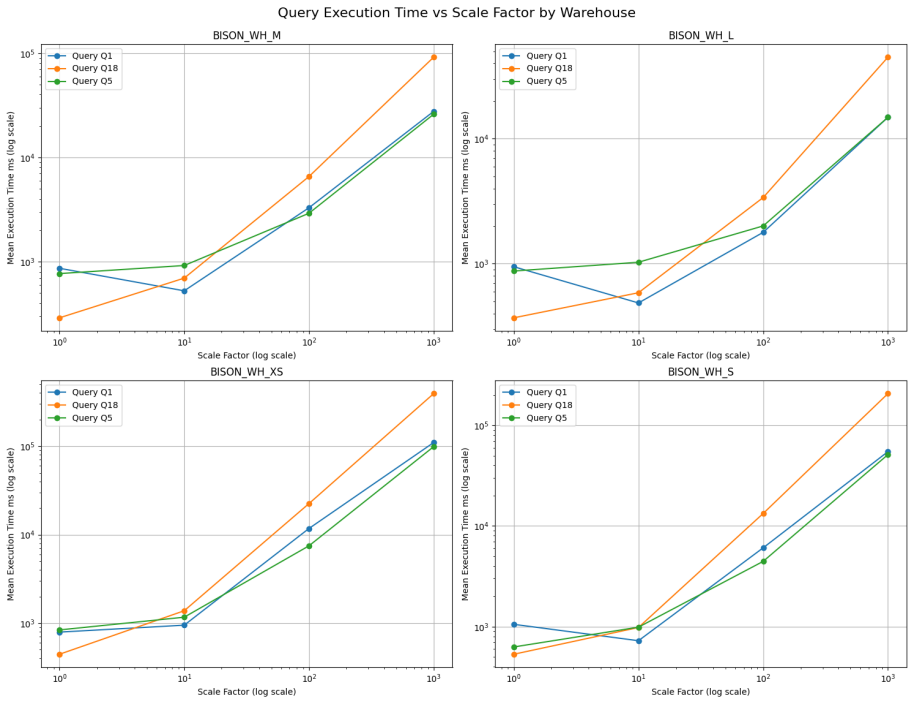

Figure 1: Log log plots of the queries under different scale factors for each warehouse size.

3.2 Naive Bayes

3.2.1 SQL

Overall Accuracy: 38.54%

I find that cleaning and exploding the training and testing sets takes $\tilde{3}5$ seconds, computing priors and likelihoods take $\tilde{1}2$ seconds and evaluating the entire cleaned testing set takes $\tilde{7}$ seconds. That's a total of 53 seconds.

3.2.2 Python UDTF

Overall Accuracy: 53.11%

UDTF 2m 46s for creating the likelihoods and counts and priors. Evaluating the test set takes 4 seconds, and is done with pure SQL.

It is curious that the accuracies are not the same, even though the mathematics of the Naive Bayes classifier should (or at least could) be the same for the two implementations. It would be interesting to look at why the mathematics differed between the two.

3.2.3 Developer Experience Comparison

The pure SQL implementation is definitely way more complicated than it ought to be. It could be greatly simplified. Disregarding the unnecessary complexity it was still very challenging to implement.

The Python UDTF implementation is much more straight forward and simple. Getting to the point where it was straight forward was not an easy journey, however. I struggled a lot with figuring out the partitioning of the UDTF.

Now that I have experience in both implementations I would definitely find it easier to implement the UDTF. Whether or not the 3x slower training time is worth the ease of implementation I will leave to the reader to decide.

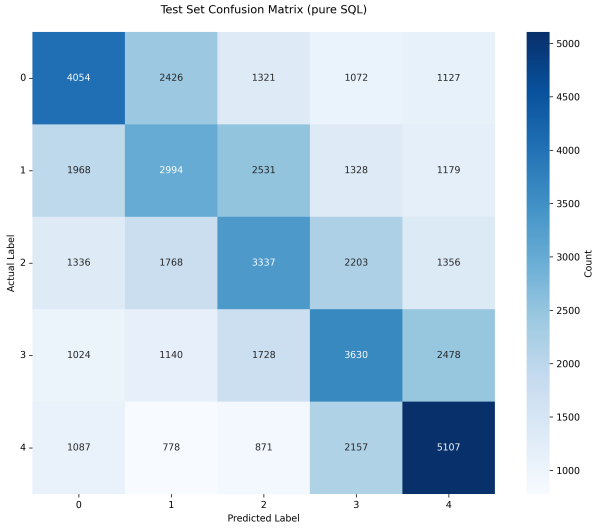

Figure 2: Confusion Matrix for the testing set using the Pure SQL implementation of Naive Bayes

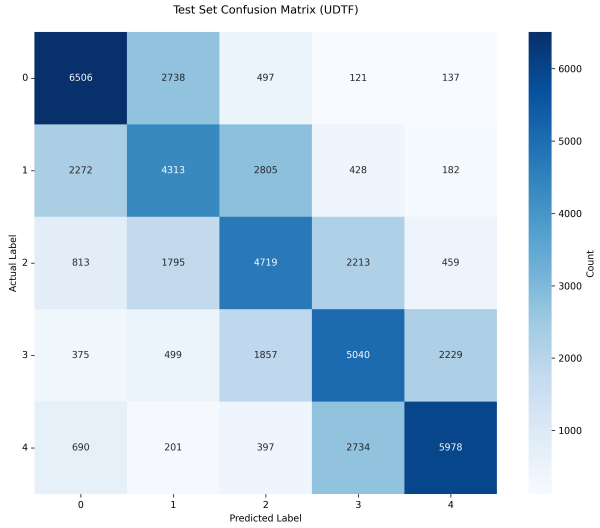

Figure 3: Confusion Matrix for the testing set using the Python UDTF implementation of Naive Bayes

4 Conclusion

I find that implementing UDTF is quite straight forward in Snowflake. I find that Query 18 of TPC-H scales the most linearly with the input size, with Query 1 and 5 first scaling linearly from scalefactor 10 upwards.

References

[1] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 215–226, New York, NY, USA, 2016. Association for Computing Machinery.